МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н. И. Лобачевского»

Институт информационных технологий математики и механики

Кафедра теоретической, компьютерной и экспериментальной механики

Направление подготовки 09.03.04 Программная инженерия

Направленность (профиль) программы бакалавриата: разработка программно-информационных систем

**Умножение плотных матриц. Элементы типа double. Блочная схема, алгоритм Кэннона.**

Исполнитель: студент **381708-2** группы

**Сегрушни Зухаир**

Руководитель: **Волокитин**.

Нижний Новгород, 2019 г.

**Оглавление**

Провозглашенные ……………………………………………………………………………………….3

Описание алгоритма и метода разрешения………………………………………………………….4

Принцип метода алгоритм Кэннона……………………………………………………………………5

Анализ производительности………………………………………………………………………..5

демонстрация …………………………………………………………………………………………….6

Приложения …….…………………………………………………………………………………………7

## 

## 

**Провозглашенные**

Умножение является одной из основных операций, используемых на матрицах. Эта операция используется практически везде.

Целью данной работы является реализация параллельных и последовательных алгоритмов, которые вычисляют произведение квадратных матриц. Следуя программе, мы должны получить другую матрицу заказов, которая является произведением двух источников, а также времени каждого алгоритма.

Тогда нужно будет реализовать последовательные и параллельные алгоритмы (Cannon). Реализуйте параллельный алгоритм, используя MPI. Кроме того, необходимо будет провести сравнение времени выполнения последовательного и параллельного алгоритмов.

**Описание алгоритма и метода разрешения**

Простейшим способом решения этой проблемы является схема умножения последовательных матриц. Он представлен тремя вложенными циклами. Сложность такого алгоритма составляет O (n ^ 3).

Double \* A, B, C;

Int size;

A = new \* double [Size];

B = new \* double [Size];

C = new \* double [Size];

for (int i = 0; i <Size; i ++)

{

A [i] = new double [Size];

B [i] = new double [Size];

C [i] = new double [Size];

}

for (int i = 0; i <Size; i ++)

for (int j = 0; j <Size; j ++)

{

A [i] [j] = rand ();

B [i] [j] = rand ();

C [i] [j] = rand ();

}

for (int i = 0; i <Size; i ++)

for (int j = 0; j <Size; j ++)

for (int k = 0; k <Size; k ++)

C [i] [j] = A [i] [k] \* B [k] [j];

**Принцип метода** **алгоритм Кэннона.**

Во-первых, создается P-число процессов выполнения. Это число должно быть целым числом квадратов. Процессы организованы в виртуальную декартовую топологию. Исходные матрицы должны иметь несколько размеров. Матрицы делятся на равное количество квадратных блоков. Блоки исходных матриц распределены среди выполнения процессов. Затем для каждой строки i в сети подмножеств блоки матрицы а перемещаются влево (i-1), а для каждого столбца j в сети подмножеств блоки матрицы в перемещаются из позиций (j-1) вверх. Затем реализуются итерации, в ходе которых блоки сначала умножаются методом трех вложенных циклов, а произведение добавляется к текущему значению результирующего блока. Затем блоки матрицы а циклируются вдоль линий сети, а блоки матрицы в циклируются до верхней части столбцов виртуальной сети. Когда все итерации завершены, результирующая матрица собирается из результирующих блоков, полученных в каждом процессе.

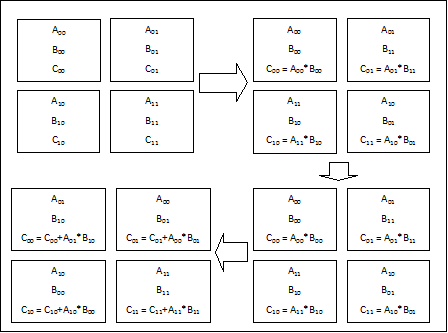
**Анализ производительности:**

алгоритм Кэннона требует Q итераций. Во время них каждый процессор умножает свои текущие блоки на исходные матрицы и генерирует текущие значения своего блока в результирующей матрице (добавляет результирующее значение к исходному). Общее количество выполненных операций будет правильным IMG_260. Результатом является значение эффективности последовательной параллельной цепи IMG_261Показатель эффективности будет выглядеть следующим образом: IMG_262.

В соответствии с алгоритмом, на стадии инициализации блоки матриц а и в перераспределяются циклическим смещением матриц по строкам и столбцам сети процессора. Сложность передачи данных зависит от топологии сети. С топологией полного графа операции могут выполняться параллельно. Стоимость переноса блоков матрицы а и в между процессорами при итерации алгоритма также может быть выполнена параллельно. Таким образом, время распределения матриц между процессами и время смещения матричных блоков равно:

IMG_263 где а-латентность, b-пропускная способность, w-размер матричного элемента. Сложность скалярного умножения строки блока матрицы а на столбец блока матрицы В может быть оценена как 2 \* (n / q) -1, количество строк и столбцов в блоках равно n / Q. сложность операции умножения блоков равна IMG_264. Добавление блоков требует IMG_265 операции. Принимая во внимание все вышеприведенные выражения, время выполнения вычислительных операций можно оценить следующим образом :  IMG_266Суммируя все полученные отношения, мы получаем общее время выполнения алгоритма Кэннона: IMG_267 где t - время выполнения скалярной операции.

**демонстрация**



**Приложения**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "mpi.h"  #include <math.h>  int allocMatrix(int\*\*\* mat, int rows, int cols) {  // Allocate rows\*cols contiguous items  int\* p = (int\*)malloc(sizeof(int\*)\* rows \* cols);  if (!p) {  return -1;  }  // Allocate row pointers  \*mat = (int\*\*)malloc(rows \* sizeof(int\*));  if (!mat) {  free(p);  return -1;  }  // Set up the pointers into the contiguous memory  for (int i = 0; i < rows; i++) {  (\*mat)[i] = &(p[i \* cols]);  }  return 0;  }  int freeMatrix(int \*\*\*mat) {  free(&((\*mat)[0][0]));  free(\*mat);  return 0;  }  void matrixMultiply(int \*\*a, int \*\*b, int rows, int cols, int \*\*\*c) {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  int val = 0;  for (int k = 0; k < rows; k++) {  val += a[i][k] \* b[k][j];  }  (\*c)[i][j] = val;  }  }  }  void printMatrix(int \*\*mat, int size) {  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  printf("%d ", mat[i][j]);  }  printf("\n");  }  }  void printMatrixFile(int \*\*mat, int size, FILE \*fp) {  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  fprintf(fp, "%d ", mat[i][j]);  }  fprintf(fp, "\n");  }  }  int main(int argc, char\* argv[]) {  MPI\_Comm cartComm;  int dim[2], period[2], reorder;  int coord[2], id;  FILE \*fp;  int \*\*A = NULL, \*\*B = NULL, \*\*C = NULL;  int \*\*localA = NULL, \*\*localB = NULL, \*\*localC = NULL;  int \*\*localARec = NULL, \*\*localBRec = NULL;  int rows = 0;  int columns;  int count = 0;  int worldSize;  int procDim;  int blockDim;  int left, right, up, down;  int bCastData[4];  // Initialize the MPI environment  MPI\_Init(&argc, &argv);  // World size  MPI\_Comm\_size(MPI\_COMM\_WORLD, &worldSize);  // Get the rank of the process  int rank;  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  if (rank == 0) {  int n;  char ch;  // Determine matrix dimensions  fp = fopen("A.txt", "r");  if (fp == NULL) {  MPI\_Abort(MPI\_COMM\_WORLD, 1);  }  while (fscanf(fp, "%d", &n) != EOF) {  ch = fgetc(fp);  if (ch == '\n') {  rows = rows + 1;  }  count++;  }  columns = count / rows;  // Check matrix and world size  if (columns != rows) {  printf("[ERROR] Matrix must be square!\n");  MPI\_Abort(MPI\_COMM\_WORLD, 2);  }  double sqroot = sqrt(double(worldSize));  if ((sqroot - floor(sqroot)) != 0) {  printf("[ERROR] Number of processes must be a perfect square!\n");  MPI\_Abort(MPI\_COMM\_WORLD, 2);  }  int intRoot = (int)sqroot;  if (columns%intRoot != 0 || rows%intRoot != 0) {  printf("[ERROR] Number of rows/columns not divisible by %d!\n", intRoot);  MPI\_Abort(MPI\_COMM\_WORLD, 3);  }  procDim = intRoot;  blockDim = columns / intRoot;  fseek(fp, 0, SEEK\_SET);  if (allocMatrix(&A, rows, columns) != 0) {  printf("[ERROR] Matrix alloc for A failed!\n");  MPI\_Abort(MPI\_COMM\_WORLD, 4);  }  if (allocMatrix(&B, rows, columns) != 0) {  printf("[ERROR] Matrix alloc for B failed!\n");  MPI\_Abort(MPI\_COMM\_WORLD, 5);  }  // Read matrix A  for (int i = 0; i < rows; i++) {  for (int j = 0; j < columns; j++) {  fscanf(fp, "%d", &n);  A[i][j] = n;  }  }  printf("A matrix:\n");  printMatrix(A, rows);  fclose(fp);  // Read matrix B  fp = fopen("B.txt", "r");  if (fp == NULL) {  return 1;  }  for (int i = 0; i < rows; i++) {  for (int j = 0; j < columns; j++) {  fscanf(fp, "%d", &n);  B[i][j] = n;  }  }  printf("B matrix:\n");  printMatrix(B, rows);  fclose(fp);  if (allocMatrix(&C, rows, columns) != 0) {  printf("[ERROR] Matrix alloc for C failed!\n");  MPI\_Abort(MPI\_COMM\_WORLD, 6);  }  bCastData[0] = procDim;  bCastData[1] = blockDim;  bCastData[2] = rows;  bCastData[3] = columns;  }  // Create 2D Cartesian grid of processes  MPI\_Bcast(&bCastData, 4, MPI\_INT, 0, MPI\_COMM\_WORLD);  procDim = bCastData[0];  blockDim = bCastData[1];  rows = bCastData[2];  columns = bCastData[3];  dim[0] = procDim; dim[1] =  procDim;  period[0] = 1; period[1] = 1;  reorder = 1;  MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dim, period, reorder, &cartComm);  // Allocate local blocks for A and B  allocMatrix(&localA,    blockDim, blockDim);  allocMatrix(&localB, blockDim, blockDim);  // Create datatype to describe the subarrays of the global array  int globalSize[2] = { rows, columns };  int localSize[2] = { blockDim, blockDim };  int starts[2] = { 0, 0 };  MPI\_Datatype type, subarrtype;  MPI\_Type\_create\_subarray(2, globalSize, localSize, starts, MPI\_ORDER\_C, MPI\_INT, &type);  MPI\_Type\_create\_resized(type, 0, blockDim \* sizeof(int), &subarrtype);  MPI\_Type\_commit(&subarrtype);  int \*globalptrA = NULL;  int \*globalptrB = NULL;  int \*globalptrC = NULL;  if (rank == 0) {  globalptrA = &(A[0][0]);  globalptrB = &(B[0][0]);  globalptrC = &(C[0][0]);  }  // Scatter the array to all processors  int\* sendCounts = (int\*)malloc(sizeof(int)\* worldSize);  int\* displacements = (int\*)malloc(sizeof(int)\* worldSize);  if (rank == 0) {  for (int i = 0; i < worldSize; i++) {  sendCounts[i] = 1;  }  int disp = 0;  for (int i = 0; i < procDim; i++) {  for (int j = 0; j < procDim; j++) {  displacements[i \* procDim + j] = disp;  disp += 1;  }  disp += (blockDim - 1)\* procDim;  }  }  MPI\_Scatterv(globalptrA, sendCounts, displacements, subarrtype, &(localA[0][0]),  rows \* columns / (worldSize), MPI\_INT,  0, MPI\_COMM\_WORLD);  MPI\_Scatterv(globalptrB, sendCounts, displacements, subarrtype, &(localB[0][0]),  rows \* columns / (worldSize), MPI\_INT,  0, MPI\_COMM\_WORLD);  if (allocMatrix(&localC, blockDim, blockDim) != 0) {  printf("[ERROR] Matrix alloc for localC in rank %d failed!\n", rank);  MPI\_Abort(MPI\_COMM\_WORLD, 7);  }  // Initial skew  MPI\_Cart\_coords(cartComm, rank, 2, coord);  MPI\_Cart\_shift(cartComm, 1, coord[0], &left, &right);  MPI\_Sendrecv\_replace(&(localA[0][0]), blockDim \* blockDim, MPI\_INT, left, 1, right, 1, cartComm, MPI\_STATUS\_IGNORE);  MPI\_Cart\_shift(cartComm, 0, coord[1], &up, &down);  MPI\_Sendrecv\_replace(&(localB[0][0]), blockDim \* blockDim, MPI\_INT, up, 1, down, 1, cartComm, MPI\_STATUS\_IGNORE);  // Init C  for (int i = 0; i < blockDim; i++) {  for (int j = 0; j < blockDim; j++) {  localC[i][j] = 0;  }  }  int\*\* multiplyRes = NULL;  if (allocMatrix(&multiplyRes, blockDim, blockDim) != 0) {  printf("[ERROR] Matrix alloc for multiplyRes in rank %d failed!\n", rank);  MPI\_Abort(MPI\_COMM\_WORLD, 8);  }  for (int k = 0; k < procDim; k++) {  matrixMultiply(localA, localB, blockDim, blockDim, &multiplyRes);  for (int i = 0; i < blockDim; i++) {  for (int j = 0; j < blockDim; j++) {  localC[i][j] += multiplyRes[i][j];  }  }  // Shift A once (left) and B once (up)  MPI\_Cart\_shift(cartComm, 1, 1, &left, &right);  MPI\_Cart\_shift(cartComm, 0, 1, &up, &down);  MPI\_Sendrecv\_replace(&(localA[0][0]), blockDim \* blockDim, MPI\_INT, left, 1, right, 1, cartComm, MPI\_STATUS\_IGNORE);  MPI\_Sendrecv\_replace(&(localB[0][0]), blockDim \* blockDim, MPI\_INT, up, 1, down, 1, cartComm, MPI\_STATUS\_IGNORE);  }  // Gather results  MPI\_Gatherv(&(localC[0][0]), rows \* columns / worldSize, MPI\_INT,  globalptrC, sendCounts, displacements, subarrtype,  0, MPI\_COMM\_WORLD);  freeMatrix(&localC);  freeMatrix(&multiplyRes);  if (rank == 0) {  printf("C is:\n");  printMatrix(C, rows);  }  // Finalize the MPI environment  MPI\_Finalize();  return 0;  } |